Parallel improved Schnorr-Euchner enumeration SE++ on shared and distributed memory systems, with and without extreme pruning

Parallel Improved Schnorr-Euchner Enumeration SE++ on Shared and Distributed Memory Systems, With and Without Extreme Pruning

Fábio Correia*
Technische Universität Darmstadt

Artur Mariano
Technische Universität Darmstadt

Alberto Proença
University of Minho

Christian Bischof
Technische Universität Darmstadt

Erik Agrell
Chalmers University of Technology

Abstract

The security of lattice-based cryptography relies on the hardness of problems based on lattices, such as the Shortest Vector Problem (SVP) and the Closest Vector Problem (CVP). This paper presents two parallel implementations for the SE++ with and without extreme pruning. The SE++ is an enumeration-based CVP-solver, which can be easily adapted to solve the SVP. We improved the SVP version of the SE++ with an optimization that avoids symmetric branches, improving its performance by a factor of ≈50%, and applied the extreme pruning technique to this improved version. The extreme pruning technique is the fastest way to compute the SVP with enumeration known to date. Our parallel implementation of the SE++ with extreme pruning targets distributed memory multi-core CPU systems, while our SE++ without extreme pruning is designed for shared memory multi-core CPU systems. These implementations address load balancing problems for optimal performance, with a master-slave mechanism on the distributed memory implementation, and specific bounds for task creation on the shared memory implementation.

The parallel implementation for the SE++ without extreme pruning scales linearly for up to 8 threads and almost linearly for 16 threads. In addition, it also achieves super-linear speedups on some instances, as the workload may be shortened, since some threads may find shorter vectors at earlier points in time, compared to the sequential implementation. Tests with our improved SE++ implementation showed that it outperforms the state of the art implementation by a factor of between 35% and 60%. Our parallel implementation of the SE++ with extreme pruning achieves linear speedups for up to 8 (working) processes and speedups of up to 13x for 16 (working) processes.

1 Introduction

Lattices are discrete subgroups of the m-dimensional Euclidean space $\mathbb{R}^m$, with a strong periodicity property. A lattice $\mathscr{L}$ generated by a basis $\mathbf{B} \in \mathbb{R}^{m \times n}$, a set of linearly independent row vectors $\mathbf{b}_1,\ldots,\mathbf{b}_n$ in $\mathbb{R}^m$, is denoted by

$$\mathscr{L}(\mathbf{B}) = \{\mathbf{x} \in \mathbb{R}^m : \mathbf{x} = \sum_{i=1}^{n} \mathbf{u}_i \mathbf{b}_i, \mathbf{u} \in \mathbb{Z}^n\}, \tag{1}$$

where n is the *rank* of the lattice. When $n = m$, the lattice is said to be of *full rank*. Lattices have a wide range of applications. These span from mathematics (e.g. geometry of numbers [9]) to computer science (e.g. integer programming [21] and lattice-based cryptography [19, 26]). The use of lattices in cryptography started in the beginning of the 80's, when the Lenstra–Lenstra–Lovász (LLL) algorithm [24] was used to break knapsack cryptosystems, and became prominent in cryptography in the mid-90's, when the first lattice-based encryption schemes were proposed (e.g. [3]).

Today, lattice-based cryptography is especially attractive because, among other reasons, it is believed to be resistant against attacks operated with quantum computers. Lattice-based cryptosystems can only

*Part of this work was performed while this author was a student at University of Minho. Email: fabio.lei.67@gmail.com

be broken when specific lattice problems can be solved in a timely manner. In this context, two lattice problems are especially relevant: the Shortest Vector Problem (SVP) and the Closest Vector Problem (CVP). The SVP consists in finding the shortest nonzero vector of the lattice, whose norm is denoted by $\lambda_1(\mathscr{L})$, or, in other words, to find $\mathbf{u} \in \mathbb{Z}^n \backslash 0$ that minimizes the Euclidean norm $\|\mathbf{B} \cdot \mathbf{u}\|$. The CVP consists in finding the closest vector of the lattice to a given target vector $\mathbf{t} \in \mathbb{R}^m$, i.e. to find $\mathbf{u} \in \mathbb{Z}^n$ minimizing $\|\mathbf{B} \cdot \mathbf{u} - \mathbf{t}\|$. Algorithms that solve these problems are usually referred to as *SVP-* and *CVP-solvers*. There is a natural connection between the SVP and CVP: the closest vector to the origin, excluding the origin itself, is the vector with norm $\lambda_1(\mathscr{L})$. From a computational perspective, the decisional variant of the CVP is known to be NP-hard [8], whereas the decisional variant of SVP is known to be NP-hard under randomized reductions [2, 16].

Both CVP- and SVP-solvers work faster on reduced lattice bases, i.e., lattices whose bases have short, nearly orthogonal vectors. The main algorithms used in practice to reduce lattices are the Lenstra-Lenstra-Lovász (LLL) and the Block Korkine-Zolotarev (BKZ) algorithms. The latter can generate bases of better or worse quality depending on an input parameter, the block-size. For higher block-sizes the quality of the output basis is better, but it also takes longer to compute it. There is a close relation between lattice reduction algorithms and SVP-solvers. BKZ, for instance, uses SVP-solvers as part of its logic, as a way to improve the quality of their output.

There are two main families of SVP- and CVP-solvers. The first is the family of sieving algorithms, i.e., probabilistic, randomized algorithms that repeatedly sieve a list of vectors, until a given stop criterion is met [4, 23]. Enumeration algorithms, on the other hand, enumerate all the possible vectors within a given search radius around the origin, and select the shortest among those [27, 20, 23].

The SVP has received much more attention than the CVP in the past decades [16]. In particular, the computational practicability of the CVP has was not very well investigated so far. While several SVP-solvers have been implemented on various computer architectures [12, 18], few open implementations of CVP-solvers are available. In particular, several CVP-solvers were proposed, as described in Section 2.2, but very few practical implementations of these solvers are available and even fewer parallel versions are known. One of the reasons why the CVP has attracted less attention than the SVP might be the lack of a public repository for the assessment of CVP-solvers, such as the SVP-challenge[1], which only covers the SVP.

On the other hand, the SVP has been continuously studied during the past decades [16]. Even though the first enumeration algorithm dates back to 1981, only recently a breakthrough technique, called extreme pruning, was proposed by Gama et al. This technique transforms enumeration into a heuristic, i.e., it is not guaranteed to find the shortest vector of the lattice, and is the fastest way to solve the SVP with enumeration known to date. Extreme pruning significantly reduces the probability of finding the shortest vector of the lattice, but the execution time of enumeration decreases in a much higher pace. One can considerably increase the probability of success by computing the extreme pruned enumeration on different bases of the same lattice. In addition, as mentioned before, some lattice basis reduction algorithms use SVP-solvers as part of their logic. An algorithm that uses the extreme pruned enumeration was proposed in [10], the BKZ 2.0. This algorithm offers the best tradeoff between quality of the output basis and execution time, thus being considered the best lattice basis reduction algorithm known to date. Therefore, better implementations of the extreme pruned enumeration is of major relevance.

Our contribution: This paper is an extension of the paper [11]. In [11], we studied the practicability of the CVP, to which end we implement and assess the performance of an enhanced version of the Schnorr-Euchner enumeration routine, described in [15], a CVP-solver that can easily be modified to solve the SVP, from here on referred to as SE++. In particular, we proposed a parallel version of this algorithm for shared-memory CPU systems, implemented with OpenMP, and we analyzed its perfor-

[1] http://www.latticechallenge.org/svp-challenge/

mance on a 16-core CPU system. In addition, we improved the SVP version of the SE++ by avoiding the computation of symmetric branches of the enumeration tree, which generate vectors with identical norms and are, therefore, irrelevant in the context of the SVP. We analyzed the performance of this improved version of the SE++ against the parallel SVP-solver proposed in [12].

In this extended version of the paper, we propose the parallelization of a variant of the improved version of the SE++, and we look at the parallelization of SE++ more holistically, this time including results with extreme pruning. In this implementation, we applied the extreme pruning technique to the SE++. This technique solves the SVP only and decreases the execution time of the SE++ considerably. Our implementation targets distributed memory multi-core CPU systems and was implemented with MPI. We also analyze the performance and scalability of this implementation.

Results: Our results show that enumeration-based CVP-solvers, whose scalability was never studied, can be parallelized at least as efficiently as enumeration-based SVP-solvers, based on a comparison of the CVP and SVP versions of our algorithm and the state of the art SVP implementation described in [12]. In particular, our parallel version of this algorithm achieves super-linear speedups in some instances on up to 8 cores and a speedup factor of 14.8x for 16 cores when solving the CVP on a 50-dimensional lattice, on a dual-socket machine with 16 physical cores. On the SVP variant of the SE++ algorithm, we improved the algorithm in such a way that it outperforms that of [12] by a factor of 35%-60%, depending on the lattice dimension, thus becoming the fastest full enumeration-based SVP-solver to date.

In addition, we implemented a scalable parallel version of the Improved SE++ with extreme pruning, for distributed memory systems. This implementation outperforms the Improved SE++ by a large margin since it avoids a huge chunk of computation by discarding branches of the enumeration that have a low probability of containing the shortest vector of the lattice. This implementation scales linearly for up to 8 processes and almost linear speedups of up to 13x for 16 working processes. Since it uses only minimal synchronization between the master process and the working processes, our implementation does not incur in a significant overheard.

Roadmap: The rest of this paper is organized as follows. Section 2 introduces the notation used and definitions. Section 2.1 overviews CVP/SVP-solvers and available implementations. Section 3 overviews the SE++ algorithm in detail, the optimization that avoids symmetric branches, its parallel implementation, and its mechanism that balances the workload among threads. Section 4 describes the enumeration with extreme pruning technique and the respective parallel implementation. Section 5 shows the results of the performance and scalability of our implementation, for both the CVP and the SVP, as well as in comparison to the implementation of the SVP-solver described in [12]. It also shows the results of the performance and scalability of the Improved SE++ with extreme pruning. Finally, Section 6 concludes the paper.

2 Notation

The Euclidean norm of a vector $\mathbf{v} \in \mathbb{R}^n$, denoted by $\|\mathbf{v}\|$, is the distance spanned from the origin of the lattice to the point given by the vector $\mathbf{v}$, i.e. $\|\mathbf{v}\| = \sqrt{\sum_{i=1}^{n} \mathbf{v}_i^2}$, where $\mathbf{v}_i$ is the ith coordinate of $\mathbf{v}$. Vectors and matrices are written in bold face, vectors are written in lower-case, and matrices in upper-case, as in vector $\mathbf{v}$ and matrix $\mathbf{M}$, and their scalar elements are denoted by $\mathbf{v}_i$ and $\mathbf{M}_{i,j}$, respectively. The absolute value of a is given by $|a|$. The lattice $\mathscr{L}$ generated by a basis $\mathbf{B}$ is denoted $\mathscr{L}(\mathbf{B})$.

2.1 Related Work

This section overviews the development of the enumeration algorithms for the SVP and CVP, in Section 2.2, and the corresponding sequential and parallel implementations, in Section 2.3. Lattice-reduction al-

gorithms, and algorithms for the approximate SVP fall out of the scope of this paper, and are not, therefore, overviewed in this section. Algorithms for the approximate CVP are briefly recapped in Section 2.2.

2.2 Algorithms

2.2.1 Exact CVP- and SVP-solvers

P. van Emde Boas showed, in 1981, that the general closest vector problem as a function of the dimension n is NP-hard [8]. The breakthrough papers in the SVP and the CVP date back to 1981, when Pohst presented an approach that examines lattice vectors that lie inside a hypersphere [27], and to 1983, when Kannan showed a different approach using a rectangular parallelepiped [20]. Extensions of these two approaches were published later on, by Fincke and Pohst, in 1985 [13], and by Kannan (following Helfrich's work [17]), in 1987 [21]. In 1994, Schnorr and Euchner proposed a significant improvement of Pohst's method [29], that was later on found to be substantially faster than Pohst's and Kannan's approaches [1]. The improvement proposed by Schnorr and Euchner was influenced by the Nearest Plane algorithm by Babai, a polynomial-time method to find vectors that are close to a given target vector [6]. In order to decrease the execution time of Schnorr and Euchner's algorithm, Schnorr and Hörner suggested a modification to enumeration in 1995 [30], called pruning. In 2010, Gama et al., based on the work of Schnorr and Hörner, proposed the enumeration with extreme pruning [14], which prunes a much higher number of nodes the enumeration tree, thus significantly reducing the execution time of the algorithm. Recently, Ghasemmehdi and Agrell showed that there are some redundant operations in the algorithm, which can be eliminated, thereby accelerating it substantially [15].

2.2.2 Approximate CVP-solvers

There are essentially two different approximate CVP-solvers: the *Nearest Plane algorithm*, developed by Babai in 1986 [6], and specific sieving algorithms. The first algorithm uses LLL to solve the approximate CVP in polynomial time, with an approximation ratio of $2(\frac{2}{\sqrt{3}})^n$, where n is the rank of the lattice. A distilled, yet precise, description of the algorithm can be found in [23].

The root of sieving algorithms dates back to 2001, when Ajtai et al. proposed a randomized algorithm that solves the exact version SVP, with very high probability [4]. This algorithm became known as AKS and it was later on extended to solve the approximate CVP [5]. It is still unclear (1) how practical this algorithm can be for the CVP and (2) if and how other sieving algorithms, such as GaussSieve [25], can be modified to solve the problem. Further improvements on the AKS were proposed by Blömer et al. [7].

Figure 1 shows a time-line and the connections between the most relevant of these publications.

2.3 Implementations

GPU and CPU parallel implementations of the Schnorr-Euchner enumeration, otherwise known as ENUM, were proposed in 2009 [18], and 2010 [12], respectively. The latter achieves almost linear speedups on a 16-core machine, for the SVP. In Section 5 we show a comparison between our implementation and that described in [12].

The fplll library includes an implementation of the Kannan–Fincke–Pohst algorithm for the SVP [28]. It is still unclear what performance levels a modified version of this algorithm can attain on CVP, since it is neither included in the fplll library nor other available implementations are known. Another

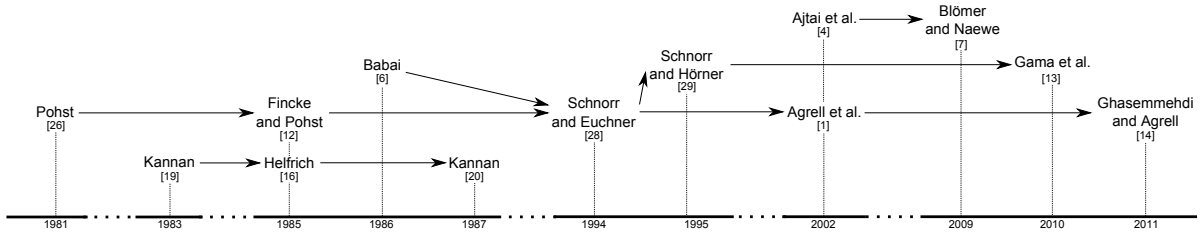


Figure 1: Timeline of the most relevant publications regarding CVP-solvers, enumeration SVP-solvers, approximate CVP-solvers, and their connections.

implementation of an enumeration process can be found in Magma[2]. However, it requires users to contribute to distribution costs (licenses start at 1000€).

The NTL library includes an implementation of the Nearest Plane algorithm for the approximate CVP (fplll includes a non-supported implementation). Comparisons with these implementations fall out of the scope of our paper, since we are only interested in performance comparisons for the SVP and CVP.

In summary, there are neither sequential nor parallel publicly available CVP-solvers, to the best of our knowledge. However, implementations of this kind are very relevant, because they permit to assess the security of lattice-based cryptosystems whose hardness is proportional to the hardness of the CVP.

An implementation of the enumeration with extreme pruning that joins the power of CPUs and GPUs was proposed in [22].

3 The SE++ Algorithm

This section provides a brief description of the closest point search algorithm, dubbed SE++, proposed by Ghasemmehdi and Agrell [15]. This algorithm is an improved version of the algorithm described by Agrell et al. called SE [1], which is based on the Schnorr-Euchner variant [29] of the Fincke-Pohst method [13].

The SE++ algorithm can be separated in two different phases: the basis pre-processing and the sphere decoding. In the pre-processing phase, the matrix that contains the basis vectors, denoted by $\mathbf{B}$, is reduced (e.g., with the BKZ or LLL algorithms). The resultant matrix $\mathbf{D}$, is transformed into a lower-triangular matrix, which we refer to as $\mathbf{G}$, with either the QR decomposition or the Cholesky decomposition (see [1] for further details). This transformation can be seen as a change of the coordinate system. The decomposition of $\mathbf{D}$ also generates an orthonormal matrix $\mathbf{Q}$. The target vector $\mathbf{r}$, i.e., the vector that we want to compute the closest vector to, is also transformed into the coordinate system of $\mathbf{G}$, i.e., $\mathbf{r'} = \mathbf{rQ}^T$. Finally, the sphere decoding is fed with the dimension of the lattice n, the transformed target vector **r'** and the inverse of $\mathbf{G}$, i.e., $\mathbf{H} = \mathbf{G}^{-1}$, which is itself a lower triangular matrix.

There are two different ways of thinking about the sphere decoding process. Mathematically, it is the process of enumerating lattice points inside a hypersphere (cf. [15] for a detailed mathematical description). Algorithmically, this can be described as a traversal of a tree, a useful view to understand the logic behind the proposed parallelization approach. In particular, it consists in a depth-first traversal on a weighted tree formed by all vectors of projections of $\mathscr{L}$ orthogonally to basis vectors. We will refer to the process of visiting a child node (decrementing i, where i denotes the depth of the node that is being analyzed at any given moment) as *moving down* and the process of visiting a parent node (incrementing i) as *moving up*.

[2] http://magma.maths.usyd.edu.au/magma/

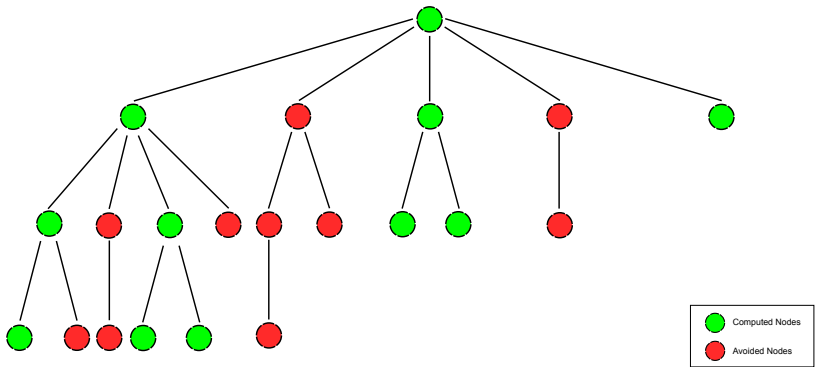


Figure 2: Representation of the symmetric subtrees whose computation can be avoided.

The algorithm iterates over all the nodes in a zigzag pattern. It starts at the root and stops when it reaches the root again. The node at depth $(i-1)$ that is being visited is determined by $\mathbf{u}_i$. The siblings of this node are visited in a zigzag pattern, based on the Schnorr-Euchner refinement [29]. Δ_i contains the step that has to be taken to visit the next node at depth $(i-1)$ and is used to update $\mathbf{u}_i$. The squared distance from the target vector $\mathbf{r}$ to the node that is being analyzed is denoted by λ_i, while C is the squared distance of $\mathbf{r}$ to the closest vector to $\mathbf{r}$ found so far. C is initialized to infinity. If $\lambda_i < C$, the algorithm will *move down*, otherwise it will *move up* again.

Whenever a leaf is reached, the values of vector $\mathbf{u}$ are saved in $\hat{\mathbf{u}}$, which represents the closest vector to $\mathbf{r}$ found so far, and C is updated, which reduces the number of nodes that still have to be visited. Although the algorithm behaves as a tree traversal, there is no physical tree (i.e. a data structure) implemented.

As proposed by Ghasemmehdi and Agrell [15], a vector $\mathbf{d}$ is used to store the starting points of the computation of the projections. The value $\mathbf{d}_i = k$ determines that, in order to compute $\mathbf{E}_{i,i}$ (see [15] for further details about matrix $\mathbf{E}$), it is only necessary to calculate the values of $\mathbf{E}_{j,i}$ for $j = k-1, k-2, \ldots, i$, thereby avoiding redundant calculations.

3.1 Implementation

3.1.1 Avoiding symmetric branches

Both the SE algorithm and its enhanced version SE++, respectively presented in [1] and [15], compute the whole enumeration tree, thereby computing several vectors that are symmetric of one another. Since the purpose of the algorithm is to find the shortest vector $\mathbf{v}$ of norm $\|\mathbf{v}\|$, it is not relevant whether $\mathbf{v}$ or $-\mathbf{v}$ is found, since $\mathbf{v}$ and **-v** have exactly the same norm. Therefore, the computation of one of these vectors can be avoided, thus reducing the number of vectors that are ultimately computed. We have incorporated this optimization in SE++, an implementation we refer to *Improved SE++*, from here on.

The ENUM algorithm avoids these computations already, by using a variable, called *last_nonzero*, which stores the largest index i of the coefficient vector $\mathbf{u}$ for which $\mathbf{u}_i \neq 0$. For example, if $\mathbf{u}_i = 0$, but its parent $\mathbf{u}_{i+1} \neq 0$, then all its subtrees have to be computed. On the other hand, there are only symmetric subtrees on nodes where $\mathbf{u}_j = 0, j = i, ..., n$. As shown in Figure 2, there are only subtrees whose computation can be avoided on the leftmost nodes of each level. Since $\mathbf{u}$ defines the subtree of each level that will be computed next, it is updated differently for nodes that contain symmetric subtrees than for nodes that do not contain them. On trees that contain symmetric subtrees, the value of $\mathbf{u}_i$ is incremented, searching only in one direction. On the other hand, on trees that do not have symmetric

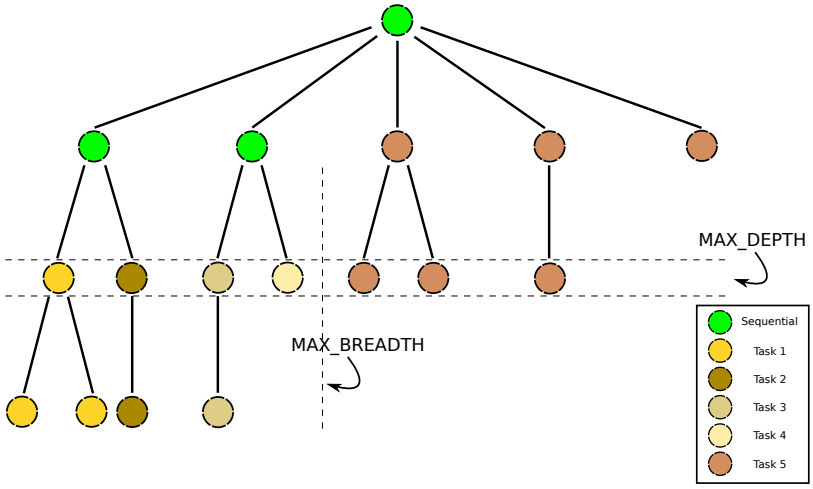


Figure 3: Map of the algorithm workflow on a tree, partitioned into tasks, according to the parameters MAX_BREADTH and MAX_DEPTH.

subtrees $\mathbf{u}_i$ is updated in a zigzag pattern, searching in both directions (positive and negative values of $\mathbf{u}_i$).

Each time the algorithm moves up on the tree and $i \geq last\_nonzero$, the variable is updated, indicating the new lowest level that contains symmetric subtrees. At the beginning of the execution, $last\_nonzero$ is initialized to 1, the index of the leaves. Due to the similarities between both algorithms, we applied this strategy to SE++ in the same way.

3.1.2 Parallelization

As previously mentioned, the workflow of the algorithm can be naturally mapped onto a tree traversal, where different branches can be computed in parallel. Figure 3 shows a partition of these branches into several tasks that can be computed in parallel, by different threads. (Very fine grained) synchronization is only used to update the best vector found at any given instant (the closest to the target vector, at a given moment), as explained below. The proposed implementation was written in C, and creates these tasks with OpenMP. Once tasks are created, they are added to a queue of tasks, and scheduled by the OpenMP run-time system among the running threads. This system also defines the order of execution of the created tasks, in run-time.

Our implementation combines a depth-first traversal with a breadth-first traversal. The work is distributed among threads in a breadth-first manner (across one or more levels), while each thread computes the work that it was assigned in a depth-first manner. First, a team of threads, whose size is set by the user, is created. Then, a number of tree nodes, based on two parameters, MAX_BREADTH and MAX_DEPTH, are computed sequentially. These two parameters also define the number and size of the tasks that are created. Once the MAX_DEPTH level is reached, a task for each of the nodes in that level is created, as also shown in Figure 3. However, when creating the task number MAX_BREADTH, i.e. $|\Delta_i|$ = MAX_BREADTH, the task entails not only the current node but also all the siblings of that node (excluding those within other tasks) and their child nodes, as shown in Task 5, in Figure 3. Once tasks are created, they are (either promptly or after some time) assigned to one of the threads within the team, by the OpenMP run-time system. As there is an implicit barrier at the end of the single region, which means that all the created tasks will be, at that point, already processed.

There is a number of problems that must be addressed in such an implementation. In first place, the tree is considerably unbalanced. $|\Delta_i|$ can be used to estimate the size of the subtree of the node that is being analyzed, because it can be seen as a relative distance between a node at depth $(i-1)$ that is being analyzed and the first vector of the same subtree that was analyzed. Therefore, $|\Delta_i|$ is used to identify heavier and lighter subtrees: the lower $|\Delta_i|$ is, the heavier the tree. As mentioned, MAX_BREADTH determines the value of $|\Delta_i|$ from which subtrees are grouped together, thus preventing the creation of too fine-grained tasks. We set MAX_BREADTH = 6, based on empirical tests presented in Section 5.

Some of the heavier subtrees need to be split in order to attain better load balancing. The maximum depth is chosen based on the number of threads on the system. In particular, the more threads, the more split the tree is. Therefore, we define $\mathsf{MAX\_DEPTH} = n - \log_2(\#Threads)$, which determines the lowest depth that is reached to split subtrees. Like MAX_BREADTH, the value for this parameter was also chosen based on empirical tests. Together, these 2 parameters represent the trade-off between the granularity and the number of created tasks.

When a thread processes a task, it computes all the nodes on the branch spanned from the root of the enumeration tree up to the root of the subtree in the task, then computing the subtree entailed by the task. The level of the subtree that was assigned, given by *i_lvl*, and the nodes that have to be recomputed, given by the vector **u_Aux**, are passed as arguments. Additionally, the value of $|\Delta_i|$ is also sent to the task, allowing to distinguish subtrees that were grouped together from single subtrees.

The recomputation of nodes that belong to previous levels of the tree allows to lower the number of memory allocations and boost performance. Instead of allocating each vector and matrix for each task, it is only necessary to allocate a much smaller vector **u_Aux** that contains the coefficients of the nodes that have to be recomputed. Therefore, each thread concurrently allocates its own (private) block of memory (a struct) for matrix **E** and vectors **u**, **y**, λ, Δ and **d** (for more about these these structures see [15]) and re-uses the same memory for the execution of all the tasks that are assigned to it. Empirical tests showed that performance can be improved by a factor of as much as 20% with this optimization.

The value of C is stored in a global variable, accessible by every thread. Threads check the value of C, which dictates the rest of the nodes that are visited by each thread. C is initialized with $1/\mathbf{H}_{1,1}$, instead of infinity, to prevent the creation of unnecessary tasks. For the same reason, $\hat{\mathbf{u}}$ is initialized with $\hat{\mathbf{u}} = (1,0,\ldots,0)$. Although these variables are shared among all the threads, only one thread updates them at a time. An OpenMP critical zone is used to manage this synchronization. Every time a thread executes the critical zone, it checks $\lambda_1 < C$ again, since other threads might update those values in the meantime.

4 Enumeration with Extreme Pruning

In 2010, Gama et al. [14] proposed a novel technique that transforms ENUM into an heuristic, which considerably reduces the number of nodes that the enumeration routine has to compute, thus significantly decreasing the execution time. However, the probability to find the shortest vector of the lattice is much lower as well, even though at a much smaller pace. This technique is called extreme pruning and can be applied to other enumeration algorithms as well, such as our Improved SE++. From here on, we refer to our Improved SE++ with extreme pruning as extreme pruned SE++.

The nodes of the tree that we have to compute are determined by a polinomial function, from here on called bounding function, based on the gaussian heuristic. The gaussian heuristic provides a good estimate of the length of the shortest vector of a lattice $\mathscr{L}$. It is calculated by $FM(\mathscr{L}) = \frac{\Gamma(n/2+1)^{1/n}}{\sqrt{\pi}} \times \mathsf{det}(\mathscr{L})^{1/n}$, where $\Gamma(x)$ is the gamma function, denoted by $\Gamma(x) = (x-1)!$, and $\mathsf{det}(\Lambda)$ is the determinant of the lattice. One can increase the probability of finding a short vector, not necessarily the shortest, by multiplying the gaussian heuristic by a factor of 1.05, as described in [22]. From here

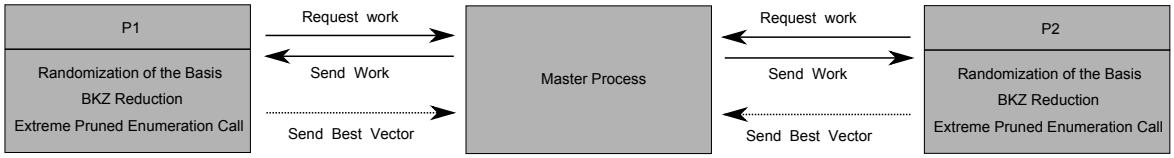


Figure 4: Communication between the master process and working processes of our extreme pruned SE++ implementation.

on, we will denote the probability of finding a short vector as success probability. The bounding function uses this value to estimate the maximum distance between a given node and the root of the tree, for each depth of the tree. We used the same bounding function as [22], which is optimal for dimension 110, and also scale for other dimensions.

As aforementioned, one call to the extreme pruned enumeration routine has only a tiny success probability of 10%, according to empirical tests presented in [22]. However, one can increase the probability of finding a short vector by performing extreme pruned enumeration calls on different bases of the same lattice. Therefore, two additional steps have to be performed before each call: (1) randomization of the basis and (2) pre-processing of the randomized basis. The pre-processing of the basis consists of reducing the randomized basis (stronger basis reductions lead to higher success probabilities) and either the QR or Cholesky decomposition.

Kuo et al. [22] also showed that 44 calls to the enumeration with extreme pruning routine guarantee a success probability over 99%. Therefore, we will use this value for the results presented in Section 5.2.

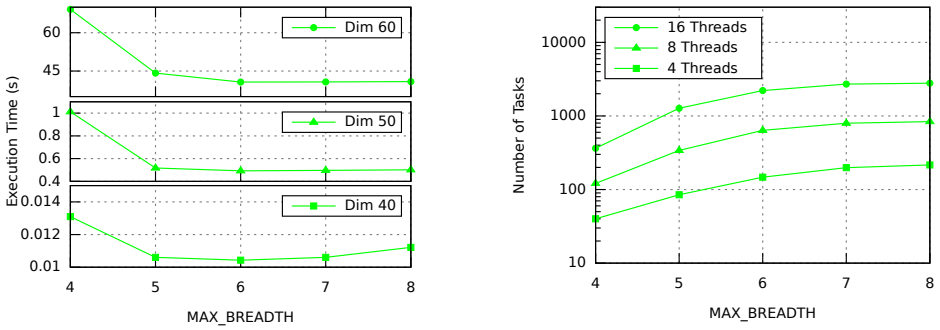
In [14], Gama et al. showed how to transform the ENUM algorithm into the enumeration with extreme pruning. One can apply the same changes to SE++ in the correspondent steps.

4.1 Parallelization

Such as the SE++, the SE++ with extreme pruning was implemented in C, as well. In addition, it distributes work among different processes with MPI. In this implementation, we also use NTL to BKZ-reduce the lattice bases.

As aforementioned, it is necessary to perform multiple calls to the extreme pruned enumeration routine (in this case an extreme pruned version of the SE++) to guarantee a high success probability. Before each call, we have to randomize the basis and pre-process the randomized basis. All of these iterations (randomization, pre-processing and extreme pruned enumeration call) are independent and can be computed simultaneously without any synchronization.

Since the execution time of each iteration depends on the quality of the randomized basis, different iterations might have different execution times. Therefore, we use an additional process (master process) to distribute iterations among working processes and to gather the best vectors found by each working process when all iterations are computed. This guarantees a more balanced work distribution, since working processes can request more iterations as soon as the previous is computed. As soon as the master process receives a request from a working process, it only sends back the number of the iteration that has to be computed, which is used as random seed in the randomization of the basis. Figure 4 shows the communication between the master process and the working processes.



(a) Execution Time (b) Number of Tasks

Figure 5: Execution time of our implementation with 16 threads solving the CVP on random lattices in dimensions 40, 50 and 60, in (a), and number of tasks created for 4, 8, and 16 threads for a lattice in dimension 50, in (b). BKZ-reduced bases with block-size 20.

5 Results

The tests were performed on a dual-socket machine with 2 Sandy Bridge Intel Xeon E5-2670 processors, each with 8 cores, and simultaneous multi-threading (SMT) technology. The machine has a total of 128 GB of RAM. The codes were written in C and compiled with the GNU g++ 4.6.1 compiler, with the -O2 optimization flag (-O3 was slightly slower than -O2). Additionally, the NTL[3] (for LLL and BKZ basis reduction) and Eigen[4] (for the QR decomposition, inverse and transpose matrix computations) libraries were also used. Although the code was written in C, the g++ compiler was required for these libraries. We have used Goldstein-Mayer bases for random lattices, available from the SVP Challenge[5], all of which were generated with seed 0. Although the execution times of the programs were fairly stable, each program was executed three times and the best sample was selected. The basis pre-processing, as described in Section 3, was not included in the time measurements for the SE++, but was included in the time measurements for the SE++ with extreme pruning.

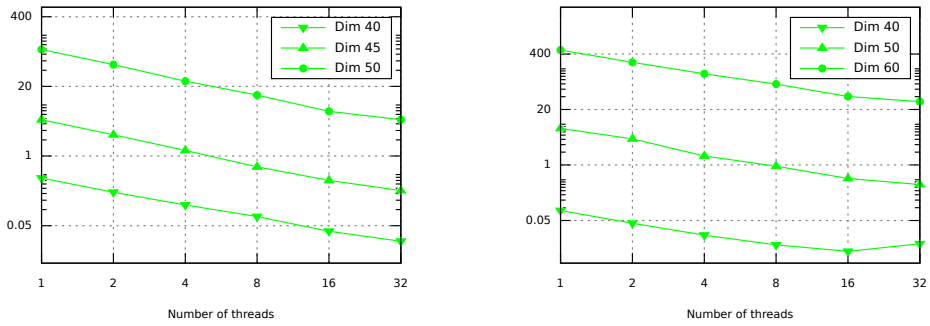
5.1 SE++ and Improved SE++

As aforementioned, in the parallel SE++ and Improved SE++, two parameters are used to prevent the creation of too many fine-grained tasks and to break down the biggest tasks into smaller tasks. The value for each of these parameters was set based on empirical tests. Several tests were performed in order to find the optimal value of MAX_BREADTH for different lattices and number of threads, for both solvers. For simplicity, Figures 5(a) and 5(b) show the results of only some of them. Different values for MAX_BREADTH were tested for both solvers in order to find its optimal value. Figure 5(a) shows the execution time for different values of MAX_BREADTH for BKZ-reduced lattices (with block-size 20) when running with 16 threads (for other number of threads the results were very similar), when solving the CVP. Figure 5(b) shows the number of tasks that are created in our parallel implementation, for 4, 8 and 16 threads, when solving the CVP. For the SVP and the Improved SE++, the number of tasks as a function of the MAX_BREADTH is similar. The higher the value of MAX_BREADTH the higher the number of tasks that are created. We set MAX_BREADTH = 6, since it was the result that revealed to be slightly better than the others, despite of creating many more tasks than MAX_BREADTH = 5. Since the

[3] http://www.shoup.net/ntl/

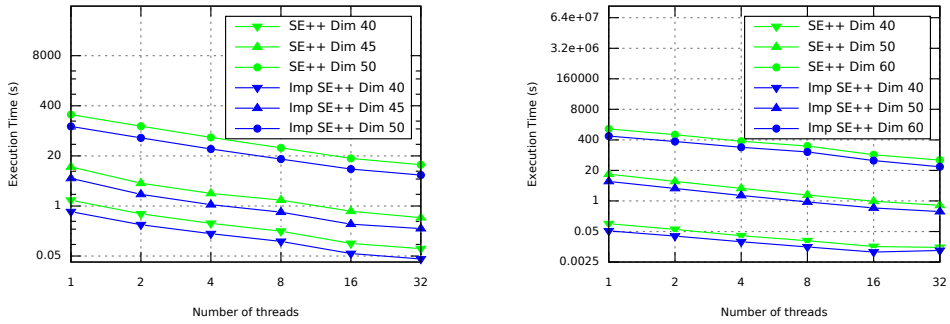
[4] http://eigen.tuxfamily.org/

[5] http://www.latticechallenge.org/svp-challenge/



(a) LLL-reduced bases (b) BKZ-reduced bases

Figure 6: Execution time of SE++ solving the CVP on random lattices in dimensions 40, 45 and 50 for LLL-reduced bases, in (a), and 40, 50 and 60, for BKZ-reduced bases, in (b), for 1-32 threads.



(a) LLL-reduced bases (b) BKZ-reduced bases

Figure 7: Execution time of our implementation solving the SVP on random lattices in dimensions 40, 45 and 50 for LLL-reduced bases, in (a), and 40, 50 and 60, for BKZ-reduced bases, in (b), for 1-32 threads.

difference between the number of created tasks is much higher than the difference between the execution time, it is possible to conclude that OpenMP's implementation of the task queue is very optimized. To choose the best values for MAX_DEPTH, the level at which tasks are created was set manually. For each level, the execution time of the tasks was registered and compared to the total execution time. To ensure linear and super-linear speedups, the execution time of the heaviest task has to be lower than $\frac{1}{\#Threads}$. To avoid creating more tasks than it is necessary to guarantee a good load balancing, MAX_DEPTH is set dynamically as $n - \log_2(\#Threads)$. With the parameters set with these values, an ideal trade-off between load balancing and granularity of the tasks is guaranteed.

We tested the SE++ and the *Improved SE++* with LLL- and BKZ-reduced bases (BKZ ran with block-size 20). For LLL-reduced bases, they were tested with lattices in dimensions 40, 45 and 50. For BKZ-reduced bases, they were tested with lattices in dimensions 40, 50 and 60, since they run much faster in BKZ-reduced bases. Figure 6(a) shows the execution time of SE++, for the CVP, running with 1-32 threads[6], with LLL-reduced bases, and Figure 6(b) shows the same tests for BKZ-reduced bases. Figures 7(a) and 7(b) show the execution time, on the same conditions, for the SVP, of SE++ and the *Improved SE++* (which includes the optimization of avoid symmetric branches), on LLL- and

[6]We used the parallel version running with a single thread as a single-core baseline, which is 5% slower than the pure-sequential version.

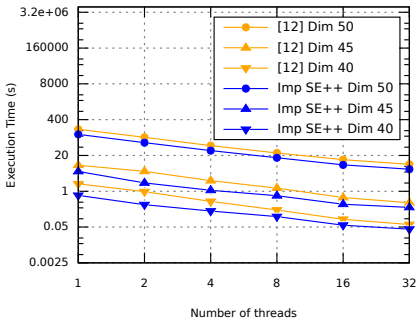


Figure 8: Execution time of our optimized implementation and the implementation in [12], for the SVP on LLL-reduced lattices.

BKZ-reduced bases, respectively.

There is a number of conclusions to be drawn. In first place, SE++ scales linearly for up to 8 threads and almost linearly for 16 threads, for both the CVP and SVP. The implementation can also benefit from the SMT technology, since the dependencies between the instructions prevent the full use of the functional units within each core. In second place, BKZ-reduced bases are much faster to compute, both for the CVP and SVP, than LLL-reduced bases. Last but not least, our implementation solves the CVP much faster than the SVP, but the results are dependent on the target vector and on the tested lattice. In our experiments, we used a vector $\mathbf{t} = \mathbf{sB}$, where $\mathbf{s}_i = 0$ for $i = 1, \ldots, n/2$ and 0.75 for $i = n/2+1, \ldots, n$, and $\mathbf{B}$ is the basis of the lattice. This vector was chosen due to not being too close to the basis vectors, but also not too far away.

A few points need to be addressed regarding the scalability of our implementation. In the first place, and as in [12], the implementation might possibly have a smaller workload than the sequential execution would have. This might occur because some threads might find, at a given point, a vector that is strictly shorter than the distance from the input vector $\mathbf{r}'$ to the $(i-1)$-dimensional layers that would be analyzed in a sequential execution. This justifies the super-linear speedups that are achieved for some cases, such as on the CVP, with a BKZ-reduced 50-dimensional lattice, using four threads.

For the remaining cases, efficiency levels of $>$90% are attained for the majority of the instances of up to 8 threads, except for lattices in dimension 40, where the workload is too small to compensate for the creation and management of more than 4 threads. With 16 threads, the scalability is lower than for up to 8 threads, presumably because of the use of two CPU sockets, which is naturally slower than the use of a single socket, due to the Non-Uniform Memory Access (NUMA) organization of the RAM. In addition, Figures 7(a) and 7(b) show that the *Improved SE++* outperforms SE++ for the SVP by a factor of $\approx$50% with similar scalability.

Figure 8 shows a comparison between the *Improved SE++* and the implementation in [12], for the SVP on two random lattices. It is possible to see that, in the general case, our implementation scales better. For the lattice in dimension 45 our approach has higher workload savings than the implementation in [12]. Both implementations are influenced by the NUMA organization of the RAM, as we can see in the case of the lattice in dimension 50. In general, with 1 thread, SE++ seems to be slower than [12], by a factor of 10% to 25%, even though it was 25% faster for the lattice in dimension 40. However, the *Improved SE++* outperforms [12] by a factor of 35% to 60%, thus becoming the fastest deterministic enumeration-based solver to date.

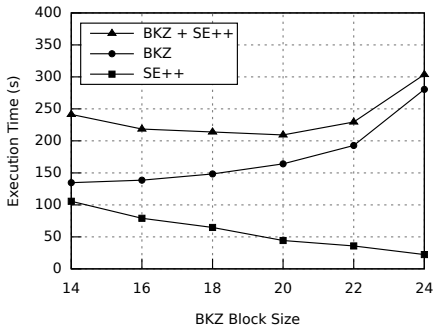


Figure 9: Execution time of the extreme pruned SE++ for different block-sizes and the corresponding time spent in the BKZ reduction and the extreme pruned SE++ call, on a lattice in dimension 80.

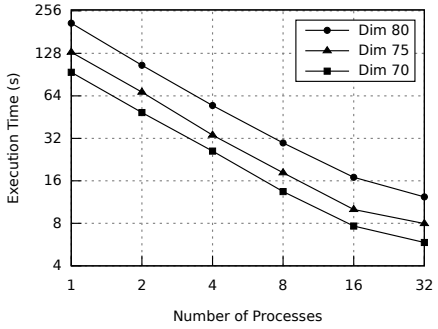


Figure 10: Execution time of our parallel extreme pruned SE++ implementation, for lattices in dimension 70, 75 and 80.

5.2 SE++ with extreme pruning

In this subsection, we analyze the performance of our implementation of the SE++ with extreme pruning. For this implementation in particular, we also included the execution time of the pre-processing of the basis in the presented results, since it takes a considerable amount of the total time to compute. We tested our implementation on lattices in dimensions 70, 75 and 80, for 1-32 working processes and one additional master process, which only sends more work to the other processes and gathers the best vectors at the end. Since the execution time of the latter is negligible, we omitted its execution time in the presented results (i.e., the mentioned number of processes only refer to the number of working processes).

Since the basis reduction takes a significant amount of time and it also influences the execution time of the extreme pruned SE++ call, we performed tests to find out the optimal value for the block-size of BKZ, used for the basis reduction. In particular, we evaluated the lattice in dimension 80, since it is the lattice in the highest dimension that was tested. Figure 9 shows that the execution time of BKZ with the block-size, but the execution time of the extreme pruned SE++ call decreases. The curve of the total execution time is a parabola with the minimum on block-size 20. Therefore, from here on, we will use block-size 20 for the remaining tests.

The first conclusion to be made is that the extreme pruning significantly decreases the execution time of the Improved SE++, as shown in Figure 10. The implementation of the SE++ with extreme pruning solves a lattice in dimension 80 in slightly more than 200s, while the Improved SE++ without extreme pruning solves a lattice in dimension 60 in almost 600 seconds, both reduced by BKZ with block-size

20.

Figure 10 also shows that our implementation scales linearly for up to 4 processes. For 8 and 16 processes, the scalability decreases due to load imbalance. This happens because the BKZ reduction and the extreme pruned SE++ call are algorithms that strongly depend on the quality of the basis. Even though the lattice is the same, we have different randomized bases on each iteration. Therefore, the execution time of BKZ and of the extreme pruned enumeration differ according to the new basis. Figure 10 also shows that the implementation benefits from SMT, since dependencies between instructions might hinder the total usage of the functional units of each core.

Our extreme pruned SE++ implementation achieves efficiency levels over 90% for up to 4 working processes and close to 90% for 8 processes. For a higher number of processes, efficiency levels decrease due to load imbalance.

6 Conclusions

This paper presents parallel implementations for the SE++ with and without extreme pruning. For the SE++ with extreme pruning, we propose an approach that targets distributed memory multi-core CPU systems, while our implementation of the SE++ without extreme pruning was developed for shared memory multi-core CPU systems.

The main characteristic of our parallel implementation for the SE++ without extreme pruning is the usage of two parameters, MAX_BREADTH and MAX_DEPTH, that ensure a good load balancing, by (1) preventing the creation of too many fine grained tasks and (2) break down the bigger tasks into smaller, yet not too small, tasks. Our approach has scales well for both the CVP and the SVP. It scales linearly for up to 8 threads and almost linearly for 16 threads, achieving speedups of up to 14.8x for 16 threads. In some instances it was possible to achieve super-linear speedups due to work load savings of the parallel implementation when running with multiple threads in comparison to running with just one thread. It takes advantage of SMT, since some dependencies between instructions impede the full usage of the functional units of the CPU. In addition, the optimization that avoids symmetric branches speeds up our implementation by a factor of $\approx$50%, which allows to outperform the state of the art implementation, presented in [12], by a factor of between 35% and 60%.

Extreme pruning is a very important technique for enumeration algorithms, since it significantly reduces their execution time. We showed that the extreme pruned enumeration technique can also be applied to other enumeration-based SVP-solvers, such as the *Improved SE++*. As shown in the results, this technique solves the SVP for lattices in much higher dimensions in less time than the implementation without extreme pruning. For instance, the non-pruned enumeration solved the SVP of BKZ-reduced basis with block-size 20 of a lattice in dimension 60 in almost 600 seconds, while the extreme pruned SE++ took slightly more than 200 seconds for a lattice in dimension 80, with the same basis reduction. We also developed a parallel approach for this technique that can easily be applied to other extreme pruned enumeration algorithms on distributed memory systems. Our implementation scales linearly for up to 4 working processes with efficiency levels of over 90%. For 8 processes, it attains efficiency levels close to 90% and speedup factors of up to 7.1x. For 16 processes, scalability is impaired because of load imbalance, attaining speedup factors of up to 13x. The reason for this load imbalance lies in the fact that the number of iterations is fixed at 44 iterations and that it is not possible to foresee the execution time of each iteration. However, for 32 working processes, the implementation keeps scaling, since dependencies between instructions hinder the full usage of the functional units of the cores.

It is possible to conclude that it is important to study the practicability and scalability of this kind of algorithms, since they solve the most relevant problems in lattice basis cryptography, the CVP, which can be solved by the SE++ without extreme pruning only, and the SVP, which can be solved by both,

the SE++ with and without extreme pruning. In addition, this kind of algorithms can also be used as a building block for other algorithms, such as the SE++, for the CVP, for Voronoi cell-based algorithms, the SE++, for the SVP, for the BKZ, and the extreme pruned SE++ for the BKZ 2.0. Therefore, by achieving high performance implementations of these algorithms they can also be used to improve the performance of the mentioned algorithms that use them as building blocks.

Acknowledgements

We thank Özgür Dagdelen and Michael Schneider, the authors of [12], for providing us with their implementation.

References

[1] Erik Agrell, Thomas Eriksson, Alexander Vardy, and Kenneth Zeger. Closest Point Search in Lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, 2002.

[2] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 99–108, New York, NY, USA, 1996. ACM.

[3] Miklós Ajtai and Cynthia Dwork. A Public-Key Cryptosystem with Worst-Case/Average-Case Equivalence. *Electronic Colloquium on Computational Complexity (ECCC)*, 3(65), 1996.

[4] Miklós Ajtai, Ravi Kumar, and D. Sivakumar. A Sieve Algorithm for the Shortest Lattice Vector Problem. In *STOC*, pages 601–610, 2001.

[5] Miklós Ajtai, Ravi Kumar, and D. Sivakumar. Sampling Short Lattice Vectors and the Closest Lattice Vector Problem. In *IEEE Conference on Computational Complexity*, pages 53–57, 2002.

[6] László Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.

[7] Johannes Blömer and Stefanie Naewe. Sampling Methods for Shortest Vectors, Closest Vectors and Successive Minima. *Theor. Comput. Sci.*, 410(18):1648–1665, April 2009.

[8] Peter van Emde Boas. Another NP-complete partition problem and the complexity of computing short vectors in a lattice. Technical Report 81-04, Mathematische Instituut, University of Amsterdam, 1981.

[9] John W. Cassels. *An Introduction to the Geometry of Numbers (Reprint)*. Classics in mathematics. Springer, 1997.

[10] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better Lattice Security Estimates. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, pages 1–20, 2011.

[11] Fabio Correia, Artur Mariano, Alberto Proença, Christian H. Bischof, and Erik Agrell. Parallel Improved Schnorr-Euchner Enumeration SE++ for the CVP and SVP. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*, pages 596–603, 2016.

[12] Özgür Dagdelen and Michael Schneider. Parallel Enumeration of Shortest Lattice Vectors. In *Euro-Par (2)*, pages 211–222, 2010.

[13] U. Fincke and M. Pohst. Improved Methods for Calculating Vectors of Short Length in a Lattice, Including a Complexity Analysis. *Mathematics of Computation*, 44:463–463, 1985.

[14] Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice Enumeration Using Extreme Pruning. In *EUROCRYPT*, pages 257–278, 2010.

[15] Arash Ghasemmehdi and Erik Agrell. Faster Recursions in Sphere Decoding. *IEEE Transactions on Information Theory*, 57(6):3530–3536, 2011.

[16] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Algorithms for the Shortest and Closest Lattice Vector Problems. In *IWCC*, pages 159–190, 2011.

[17] Bettina Helfrich. Algorithms to Construct Minkowski Reduced an Hermite Reduced Lattice Bases. *Theor. Comput. Sci.*, 41:125–139, 1985.

[18] Jens Hermans, Michael Schneider, Johannes Buchmann, Frederik Vercauteren, and Bart Preneel. Parallel Shortest Lattice Vector Enumeration on Graphics Cards. In *AFRICACRYPT*, pages 52–68, 2010.

[19] Antoine Joux and Jacques Stern. Lattice Reduction: A Toolbox for the Cryptanalyst. *J. Cryptology*, 11(3):161–185, 1998.

[20] Ravi Kannan. Improved Algorithms for Integer Programming and Related Lattice Problems. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 193–206, New York, NY, USA, 1983. ACM.

[21] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Math. Oper. Res.*, 12(3):415–440, August 1987.

[22] Po-Chun Kuo, Michael Schneider, Özgür Dagdelen, Jan Reichelt, Johannes A. Buchmann, Chen-Mou Cheng, and Bo-Yin Yang. Extreme Enumeration on GPU and in Clouds - How Many Dollars You Need to Break SVP Challenges -. In *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, pages 176–191, 2011.

[23] Thijs Laarhoven, Joop van de Pol, and Benne de Weger. Solving hard lattice problems and the security of lattice-based cryptosystems. Cryptology ePrint Archive, Report 2012/533, 2012.

[24] A.K. Lenstra, H.W.jun. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.

[25] Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. *Electronic Colloquium on Computational Complexity (ECCC)*, 16:65, 2009.

[26] A. M. Odlyzko. The Rise and Fall of Knapsack Cryptosystems. In *In Cryptology and Computational Number Theory*, pages 75–88. A.M.S, 1990.

[27] Michael Pohst. On the Computation of Lattice Vectors of Minimal Length, Successive Minima and Reduced Bases with Applications. *SIGSAM Bull.*, 15(1):37–44, February 1981.

[28] Xavier Pujol and Damien Stehlé. Rigorous and efficient short lattice vectors enumeration. In *ASIACRYPT*, pages 390–405, 2008.

[29] Claus-Peter Schnorr and M. Euchner. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. *Math. Program.*, 66:181–199, 1994.

[30] Claus-Peter Schnorr and Horst Helmut Hörner. Attacking the Chor-Rivest Cryptosystem by Improved Lattice Reduction. In *Advances in Cryptology - EUROCRYPT '95, International Conference on the Theory and Application of Cryptographic Techniques, Saint-Malo, France, May 21-25, 1995, Proceeding*, pages 1–12, 1995.